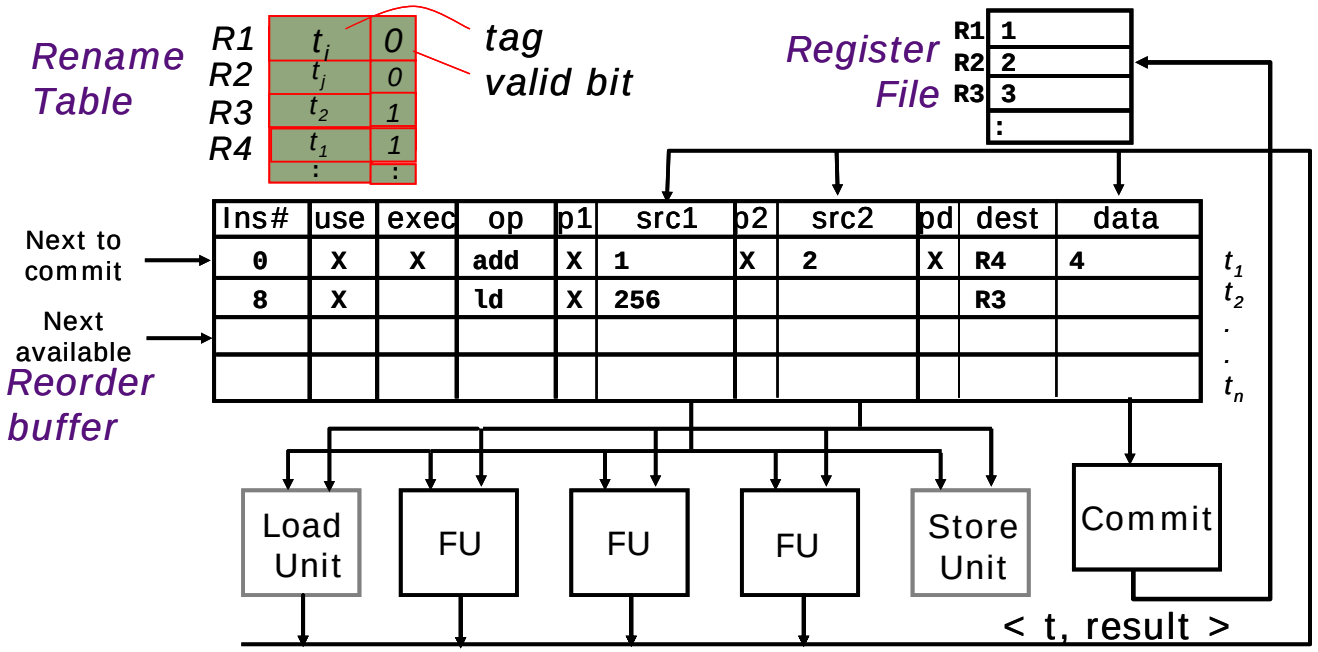
Advanced Superscalar Microprocessors

Joel Emer
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Based on the material prepared by
Krste Asanovic and Arvind

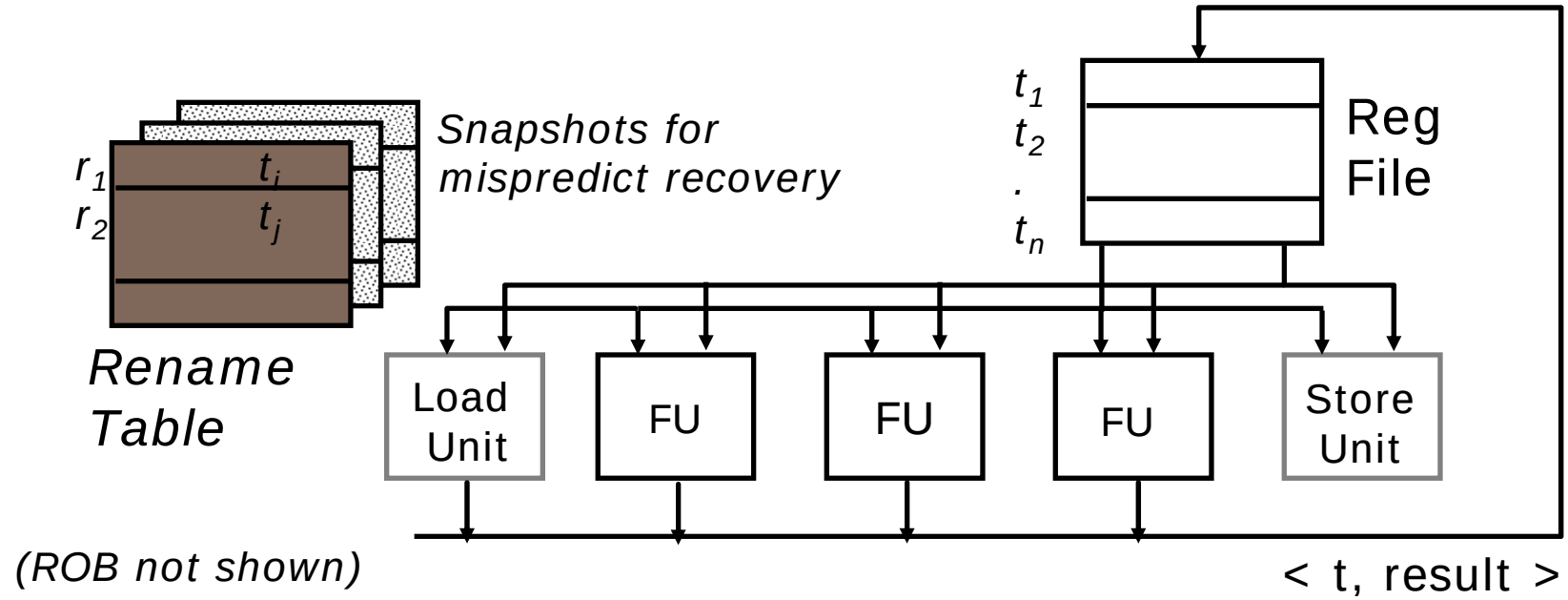
O-o-O Execution with ROB



Basic Operation:

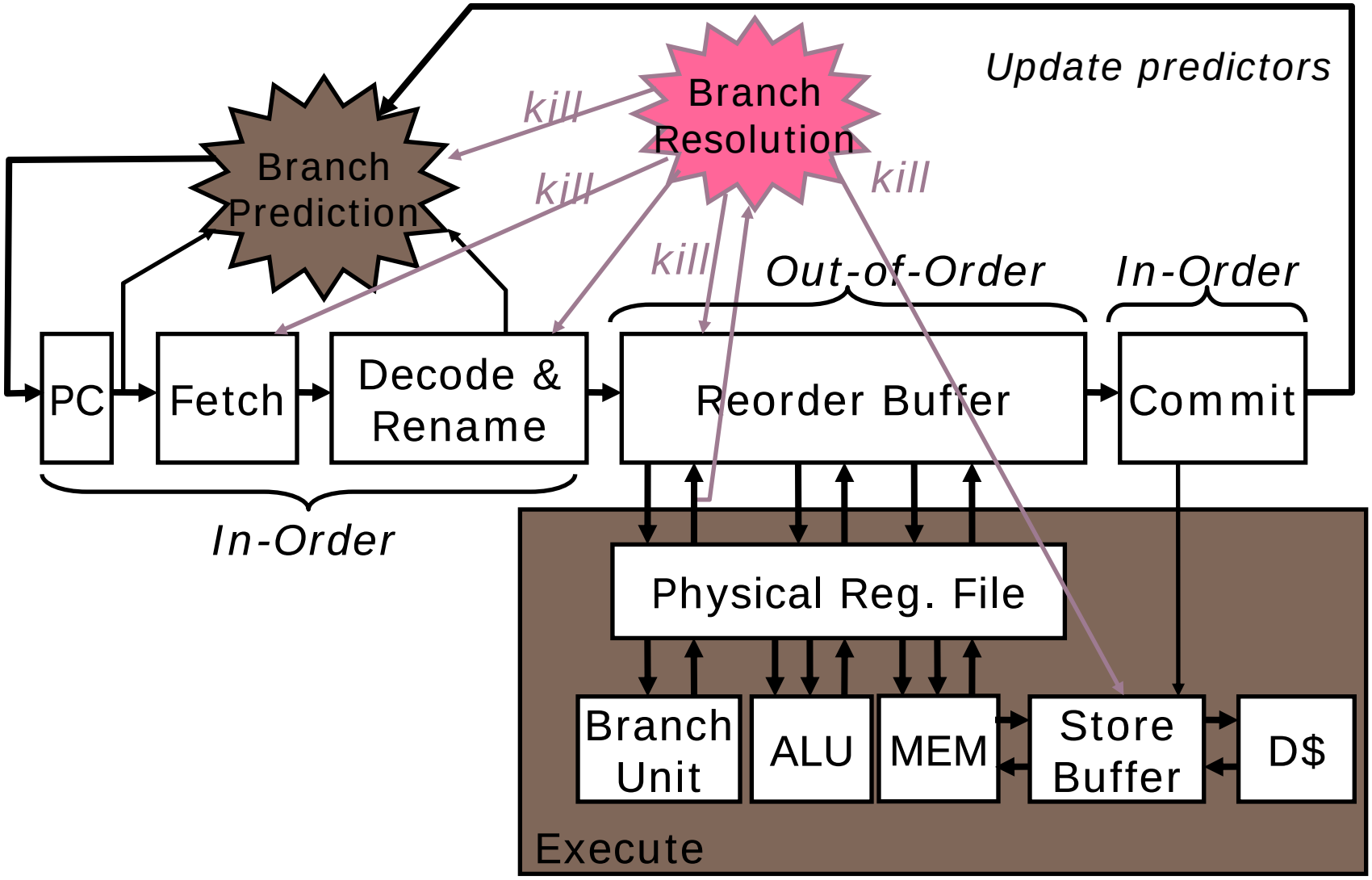
- Enter op and tag or data (if known) for each source
- Replace tag with data as it becomes available
- Issue instruction when all sources are available
- Save dest data when operation finishes
- Commit saved dest data when instruction commits

Unified Physical Register File
(MIPS R10K, Alpha 21264, Pentium 4)



- One regfile for both *committed* and *speculative* values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates reg. busy bits on instructions in ROB (assoc. search)
- Snapshots of rename table taken at every branch to recover mispredicts
- On exception, renaming undone in reverse order of issue *(MIPS R10000)*

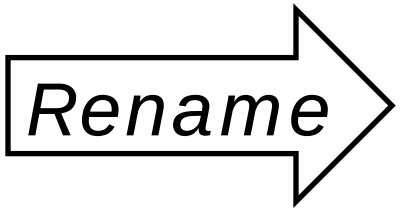
Speculative & Out-of-Order Execution



Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

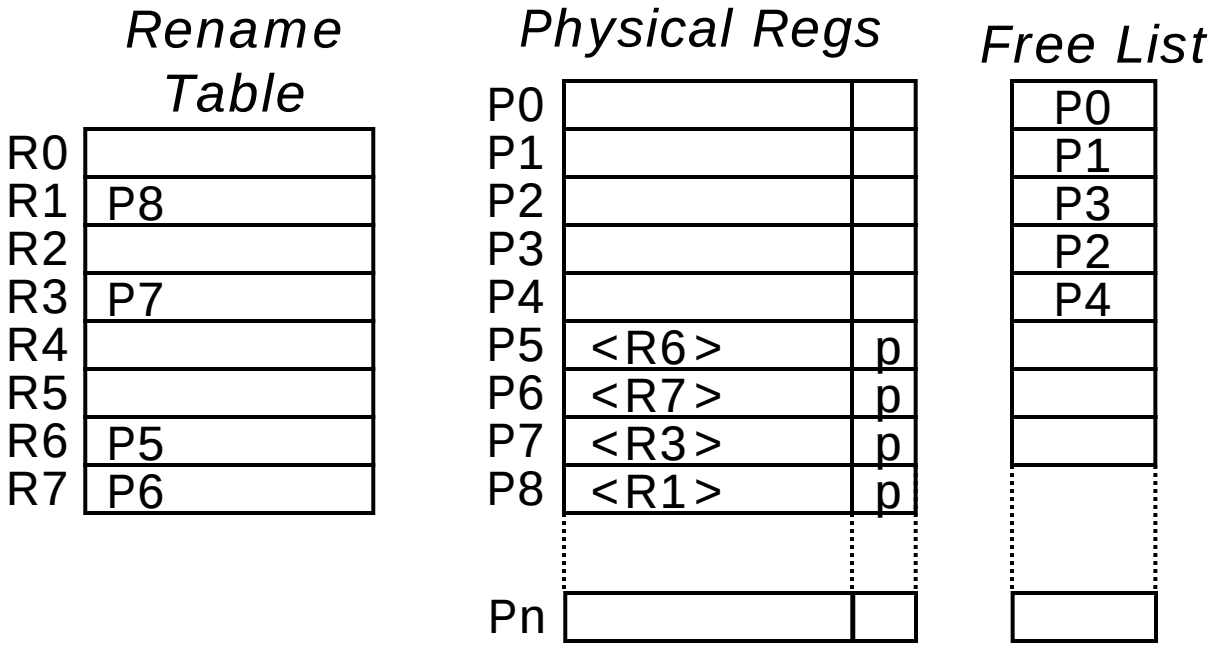
```
ld r1, (r3)                ld P1, (Px)
add r3, r1, #4             add P2, P1, #4
sub r6, r7, r9             sub P3, Py, Pz
add r3, r3, r6             add P4, P2, P3
ld r6, (r1)                ld P5, (P1)
add r6, r6, r3             add P6, P5, P4
st r6, (r1)                st P6, (P1)
ld r6, (r11)               ld P7, (Pw)
```



When can we reuse a physical register?

When next write of same architectural register commits

Physical Register Management



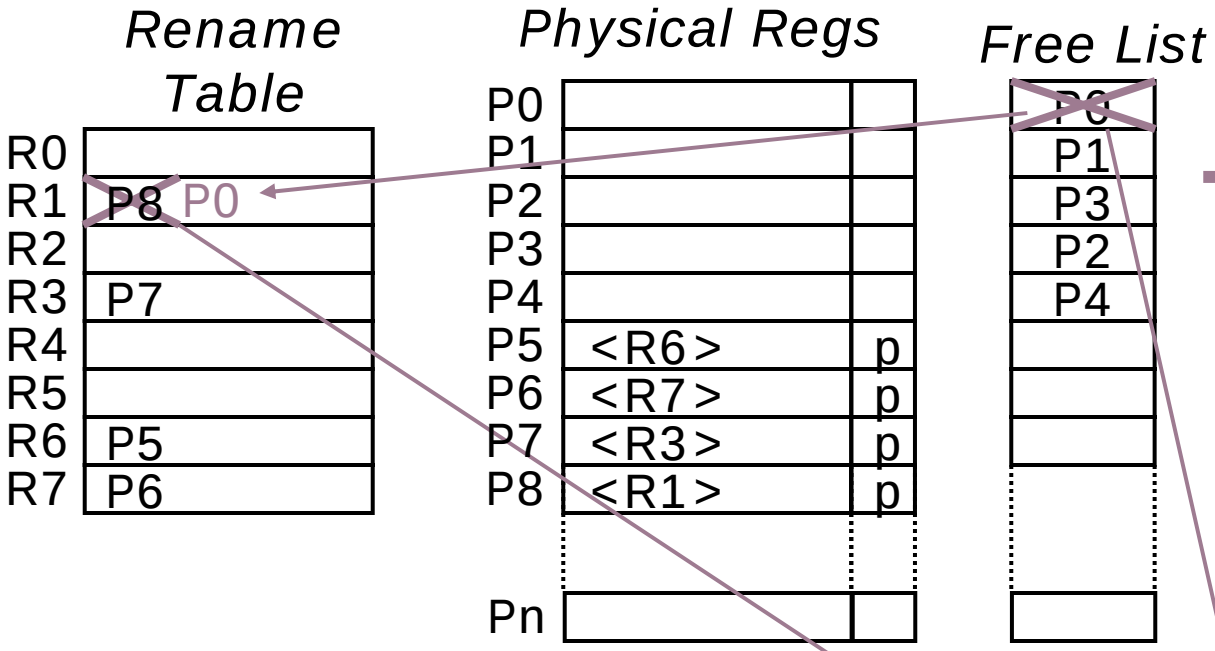

```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management

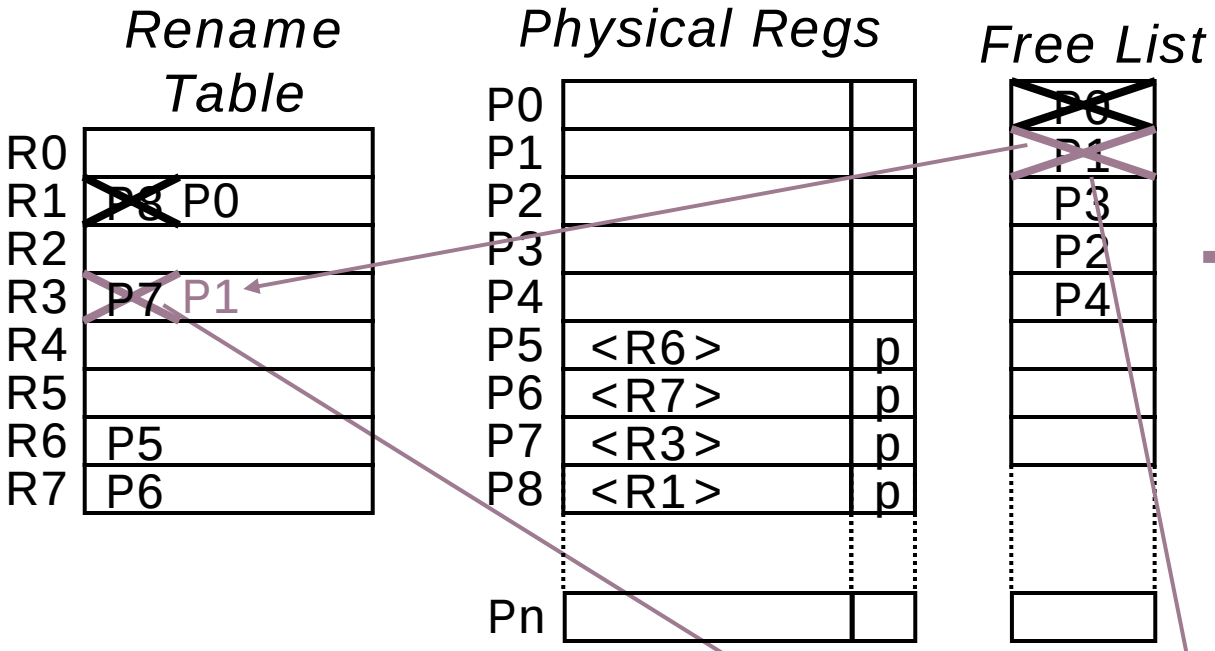


ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0

Physical Register Management

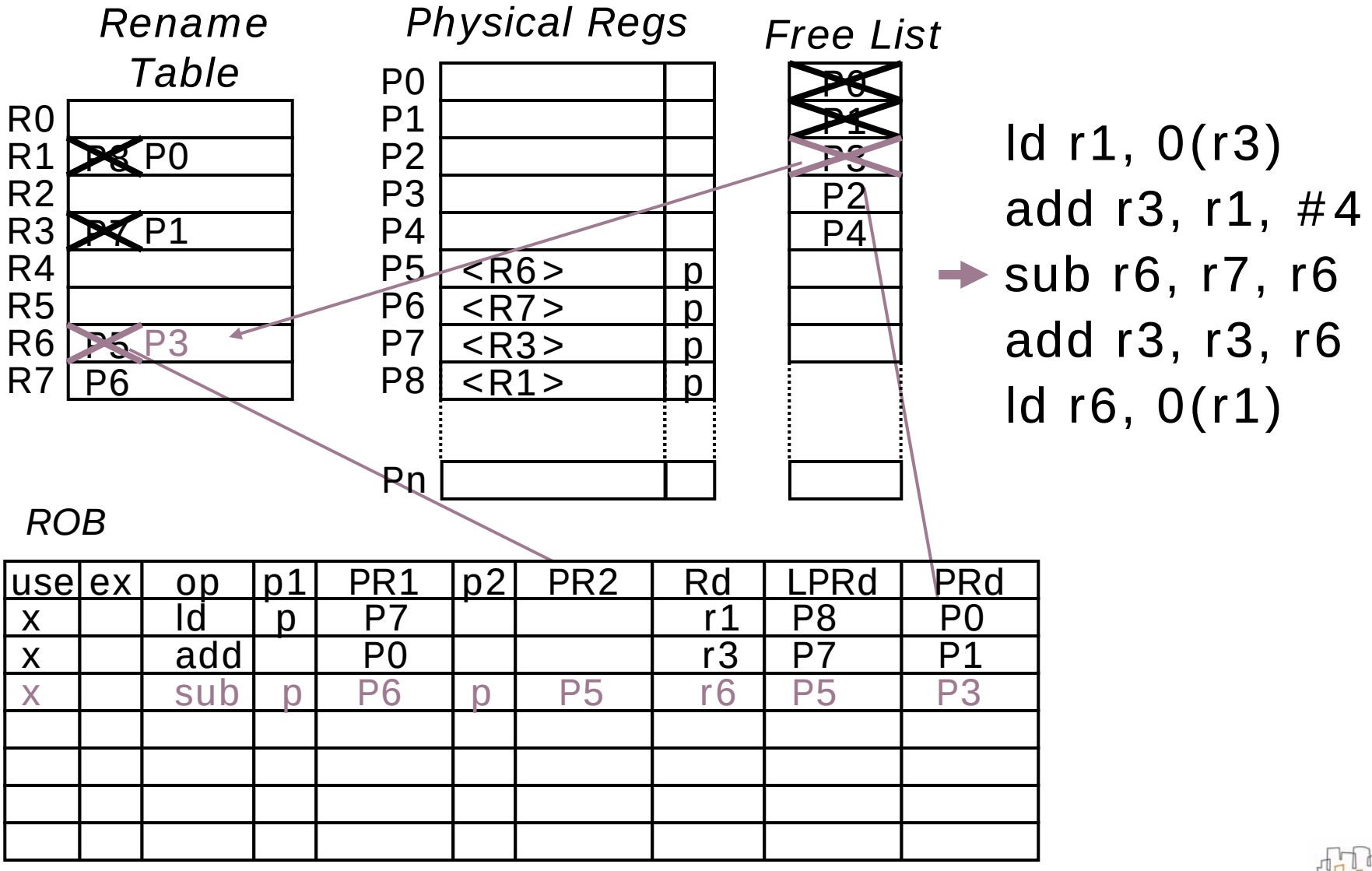



```
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
```

ROB

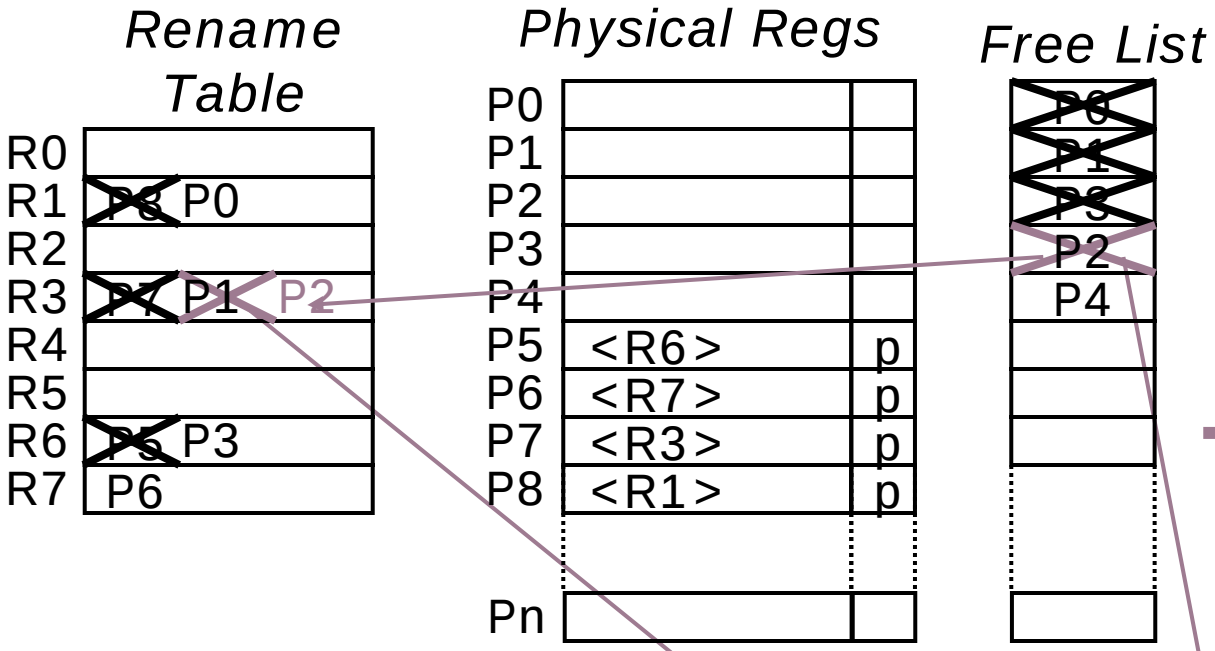
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1

Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3

Physical Register Management

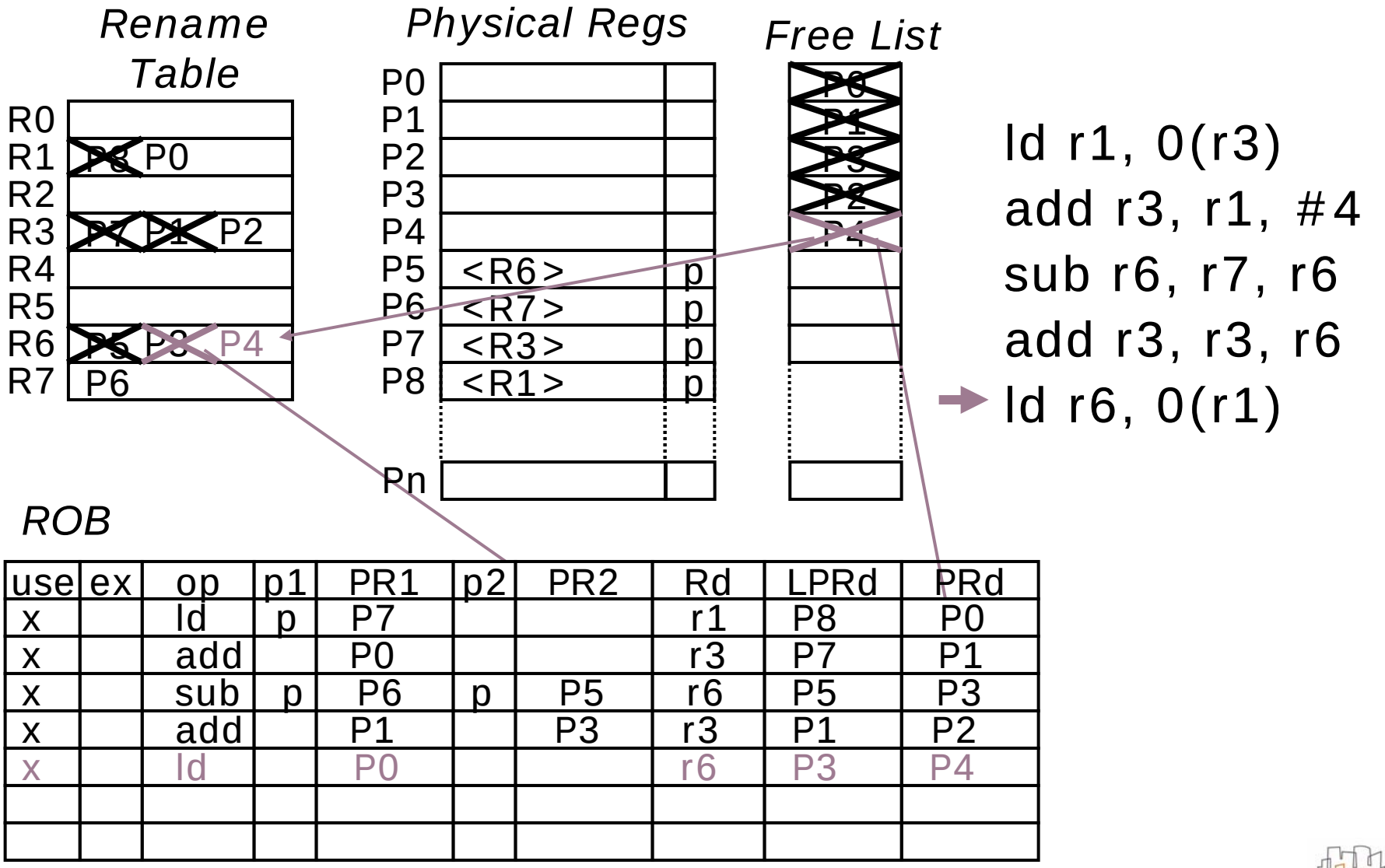


ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
→ add r3, r3, r6
ld r6, 0(r1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2

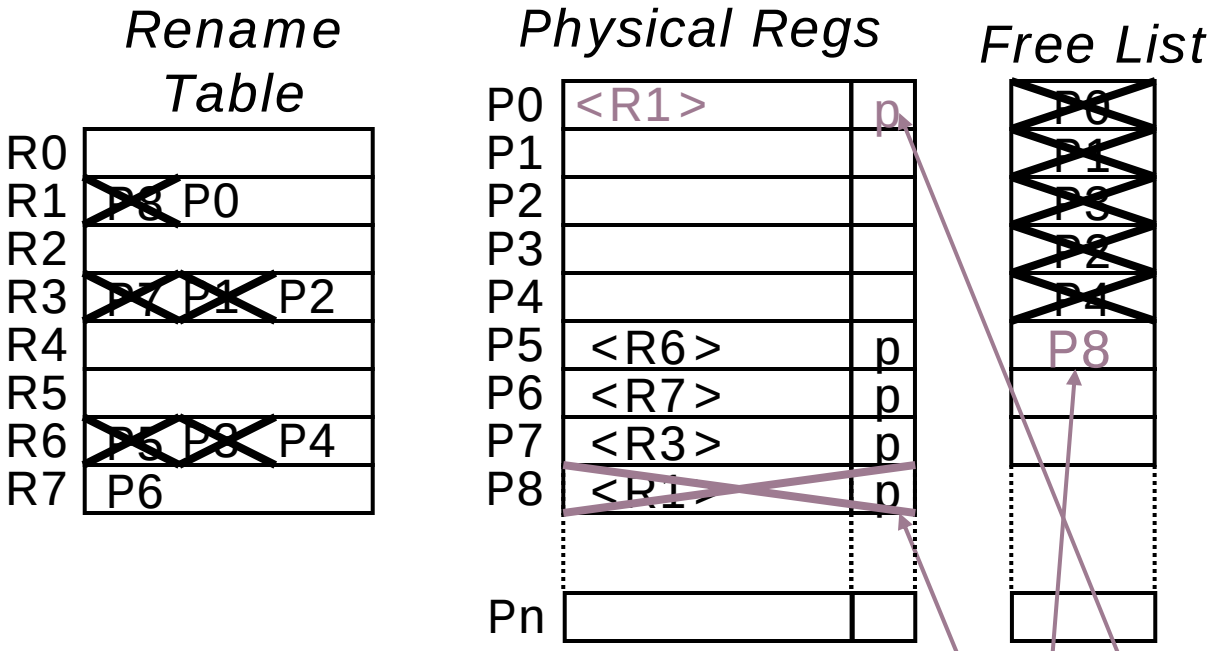
Physical Register Management



ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			r1	P8	P0
x		add		P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld		P0			r6	P3	P4

Physical Register Management



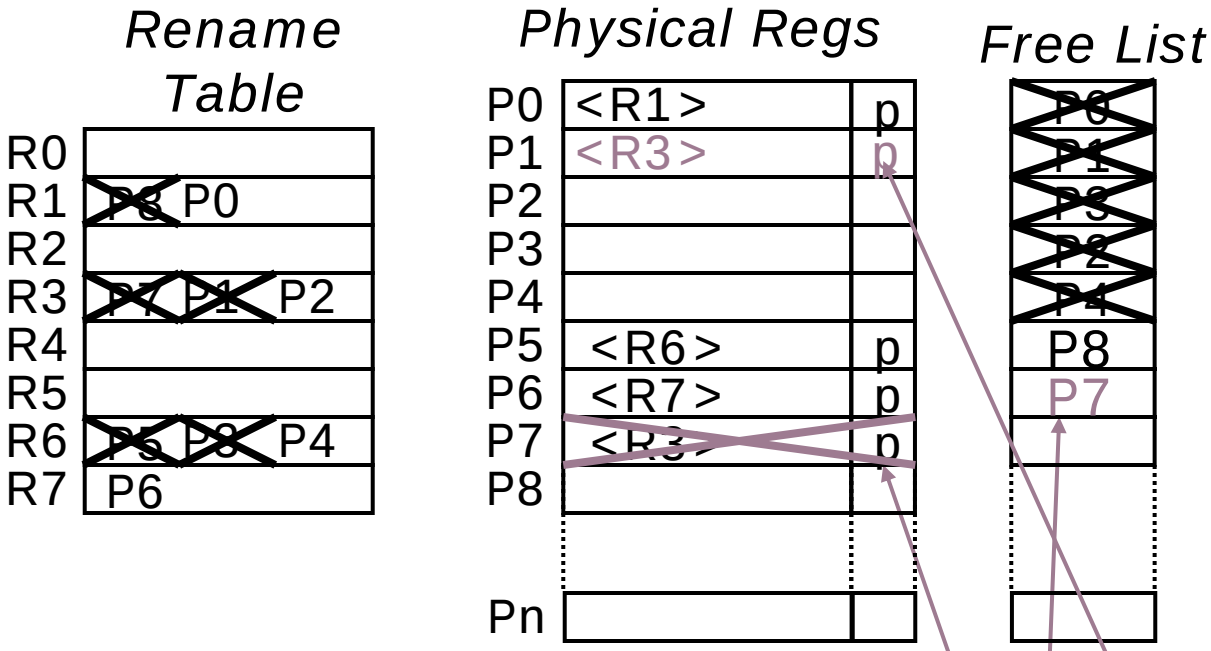
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			r1	P8	P0
x		add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add		P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

Execute & Commit

Physical Register Management



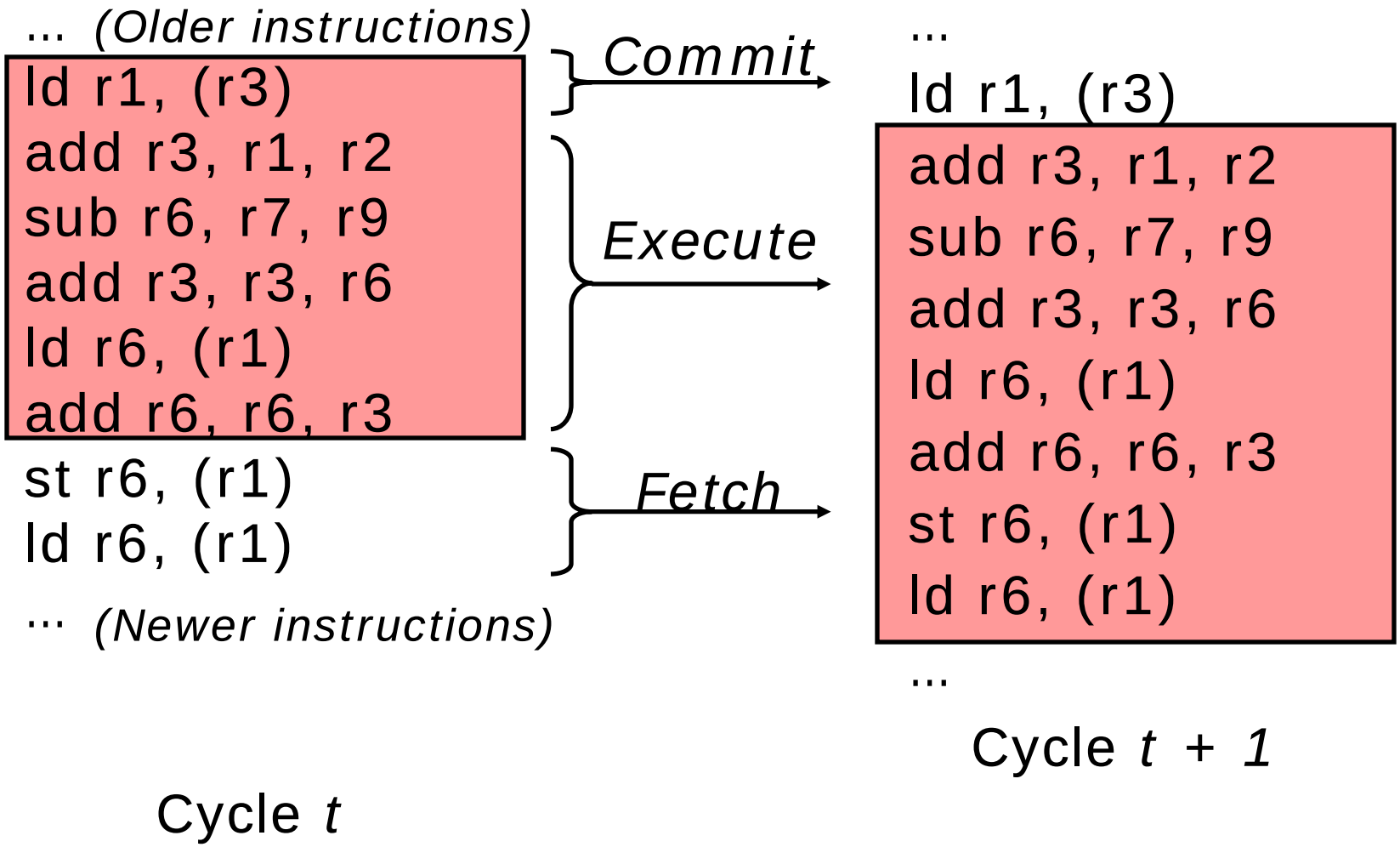
ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			r1	P8	P0
x	x	add	p	P0			r3	P7	P1
x		sub	p	P6	p	P5	r6	P5	P3
x		add	p	P1		P3	r3	P1	P2
x		ld	p	P0			r6	P3	P4

Execute & Commit

Reorder Buffer Holds Active Instruction Window



Issue Timing

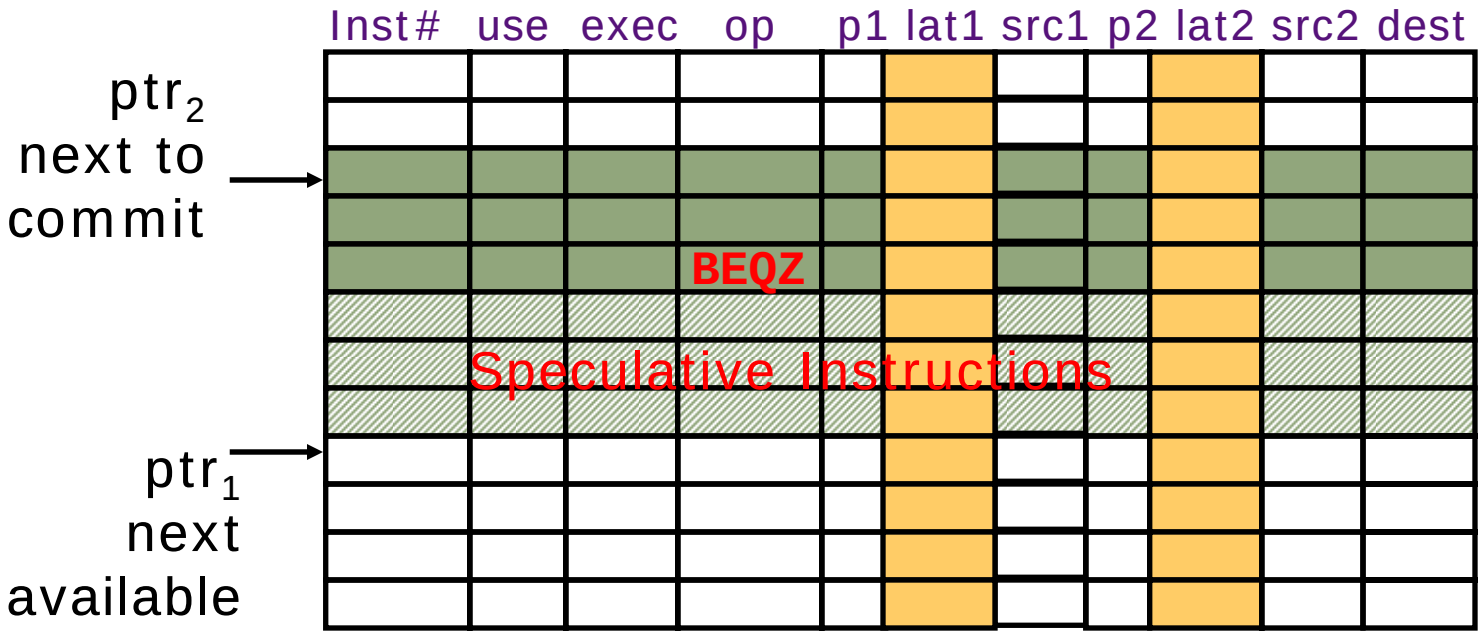
i1	Add R1,R1,#1	$Issue_1$	$Execute_1$		
i2	Sub R1,R1,#1			$Issue_2$	$Execute_2$

How can we issue earlier?

i1	Add R1,R1,#1	$Issue_1$	$Execute_1$		
i2	Sub R1,R1,#1		$Issue_2$	$Execute_2$	

What makes this schedule fail?

Issue Queue with latency prediction

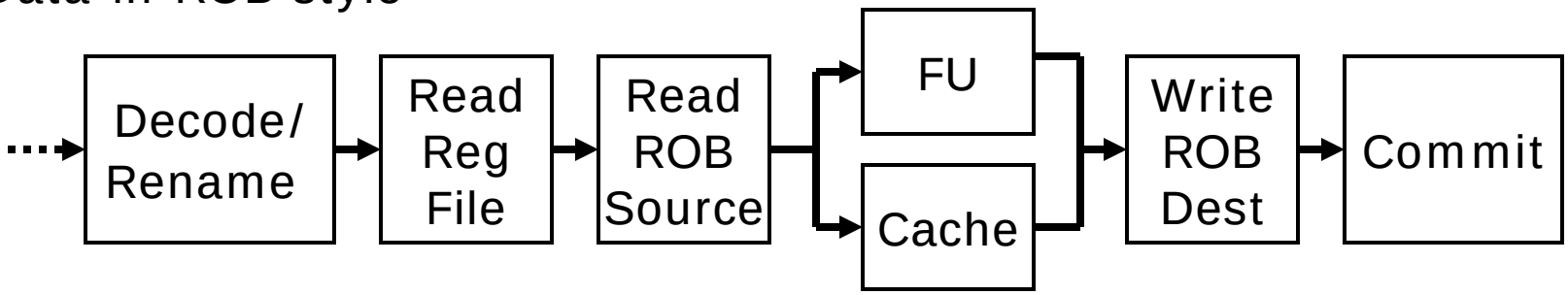


Issue Queue (Reorder buffer)

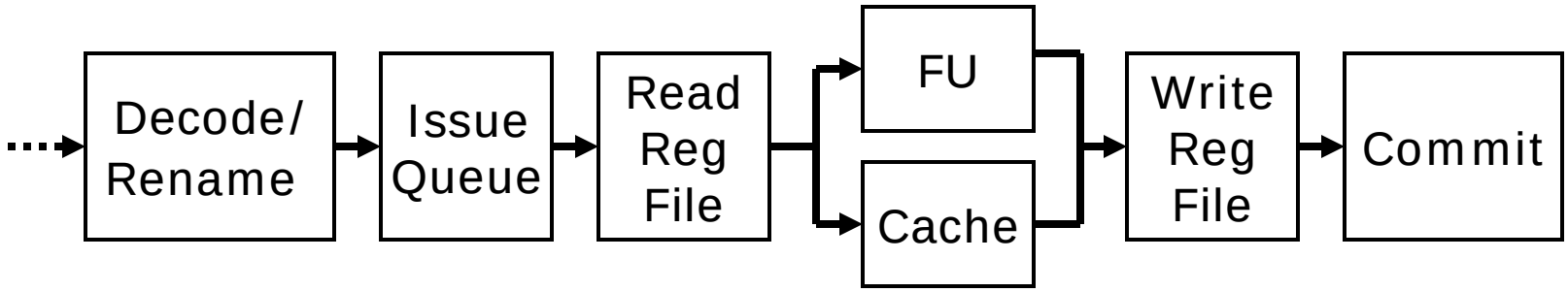
- Fixed latency: latency included in queue entry ('bypassed')
- Predicted latency: latency included in queue entry (speculated)
- Variable latency: wait for completion signal (stall)

Data-in-ROB vs. Single Register File

Data-in-ROB style



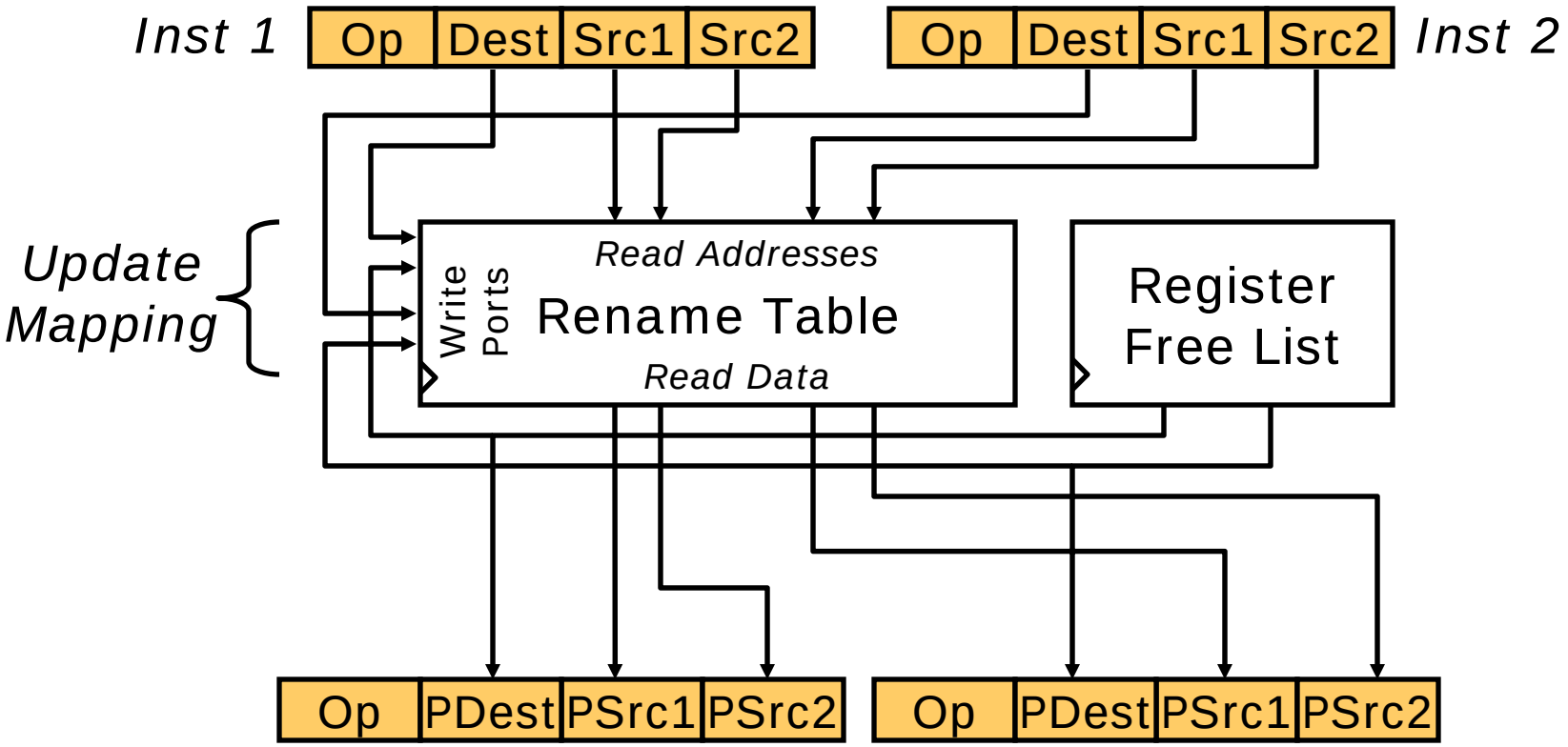
Single-register-file style



How does issue speculation differ?

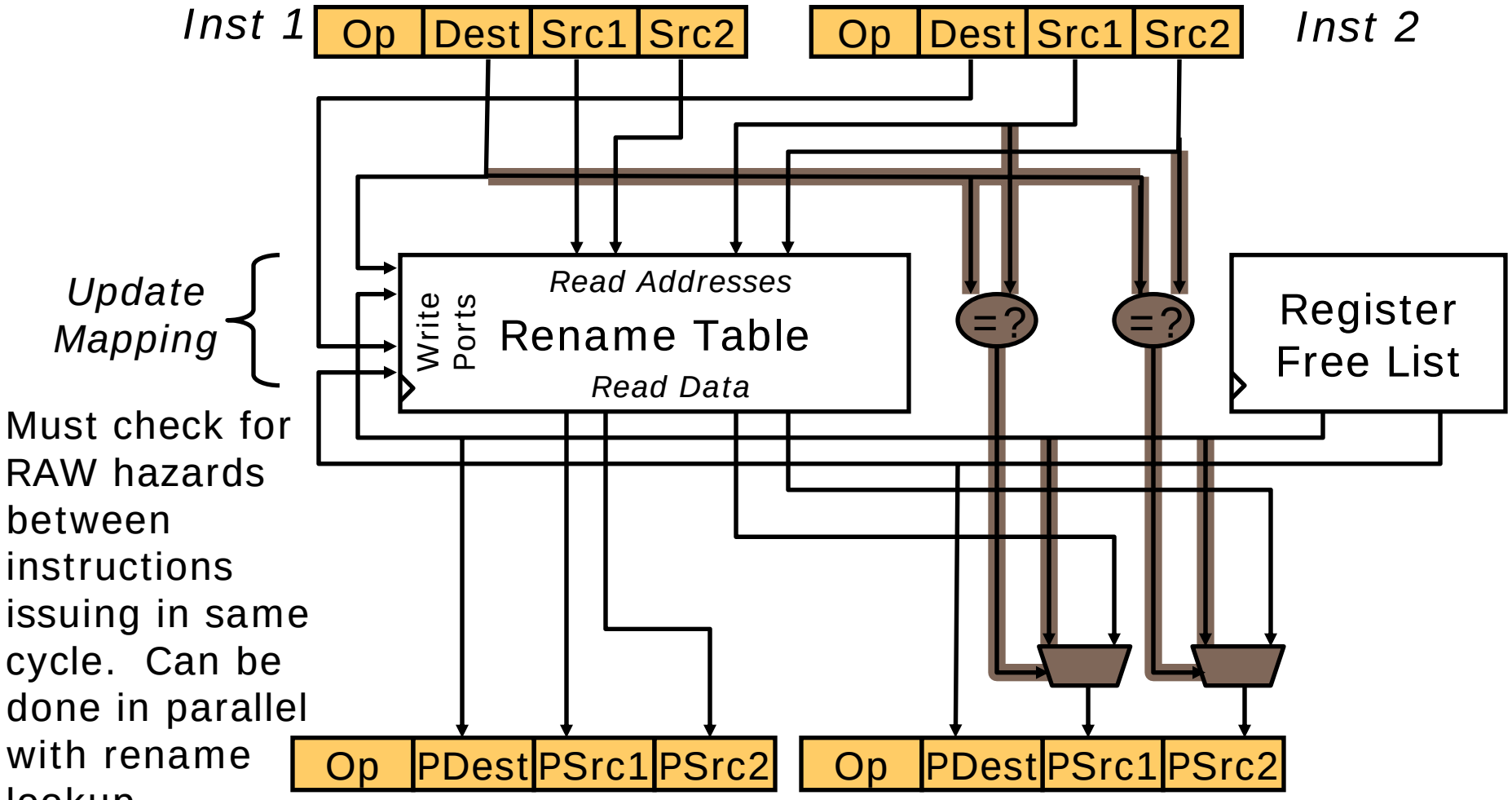
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



Does this work?

Superscalar Register Renaming



Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

MIPS R10K renames 4 serially-RAW-dependent insts/cycle)

CSAIL

Five-minute break to stretch your legs

Memory Dependencies

```
st r1, (r2)
ld r3, (r4)
```

When can we execute the load?

Speculative Loads / Stores

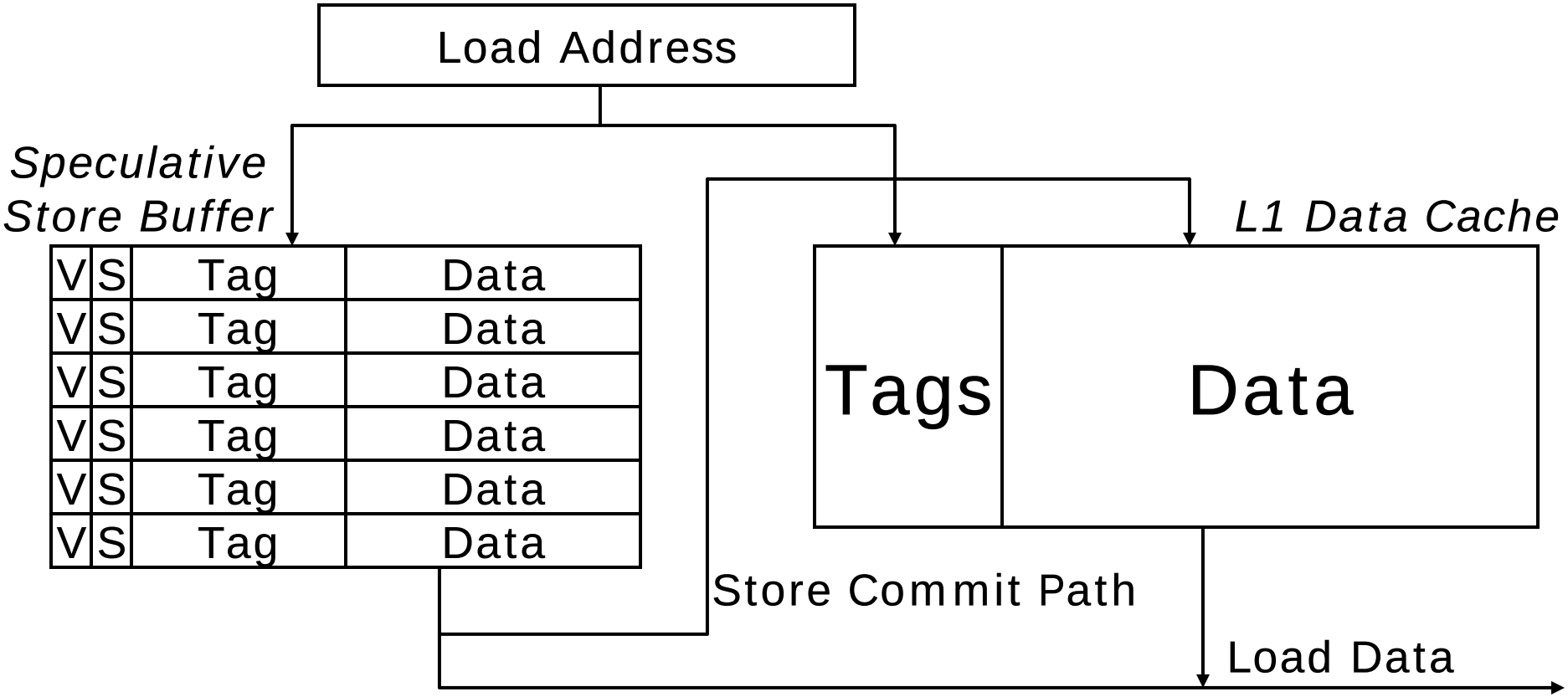
Just like register updates, stores should not modify the memory until after the instruction is committed

->store buffer entry must carry a speculation bit and the tag of the corresponding store instruction

- If the instruction is committed, the speculation bit of the corresponding store buffer entry is cleared, and store is written to cache
- If the instruction is killed, the corresponding store buffer entry is freed

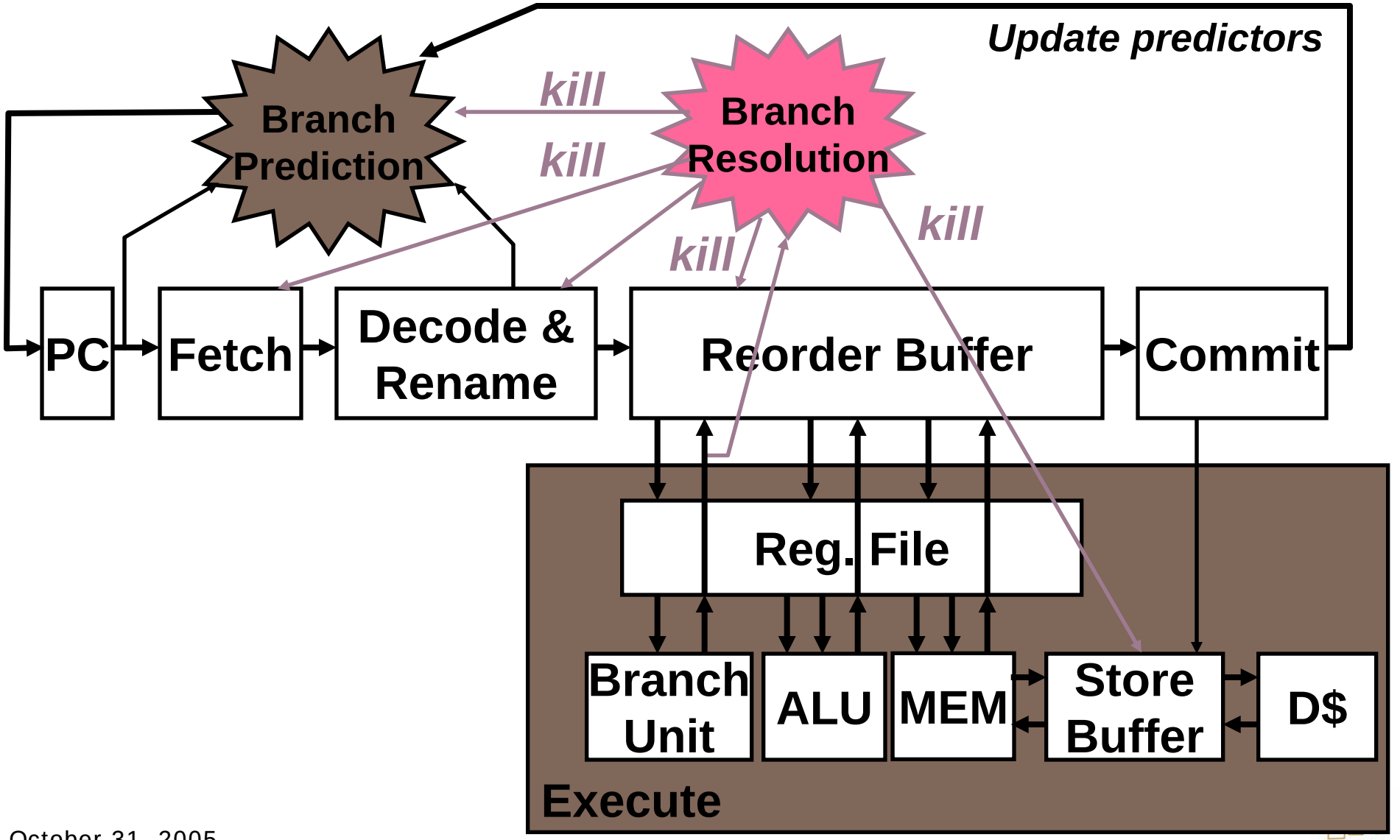
Loads work normally -- “older” store buffer entries needs to be searched before accessing the memory or the cache

Load Path



- Hit in speculative store buffer has priority over hit in data cache
- Hit to newer store has priority over hits to older stores in speculative store buffer

Datapath: Branch Prediction and Speculative Execution



In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Stores held in store buffer until commit

Conservative O-o-O Load Execution

```
st r1, (r2)
ld r3, (r4)
```

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and r4 != r2
- Each load address compared with addresses of all previous uncommitted stores *(can use partial conservative check i.e., bottom 12 bits of address)*
- Don't execute load if any previous store address not known

(MIPS R10K, 16 entry address queue)

Address Speculation

```
st r1, (r2)
ld r3, (r4)
```

- Guess that r4 != r2
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find r4==r2, squash load and *all* following instructions

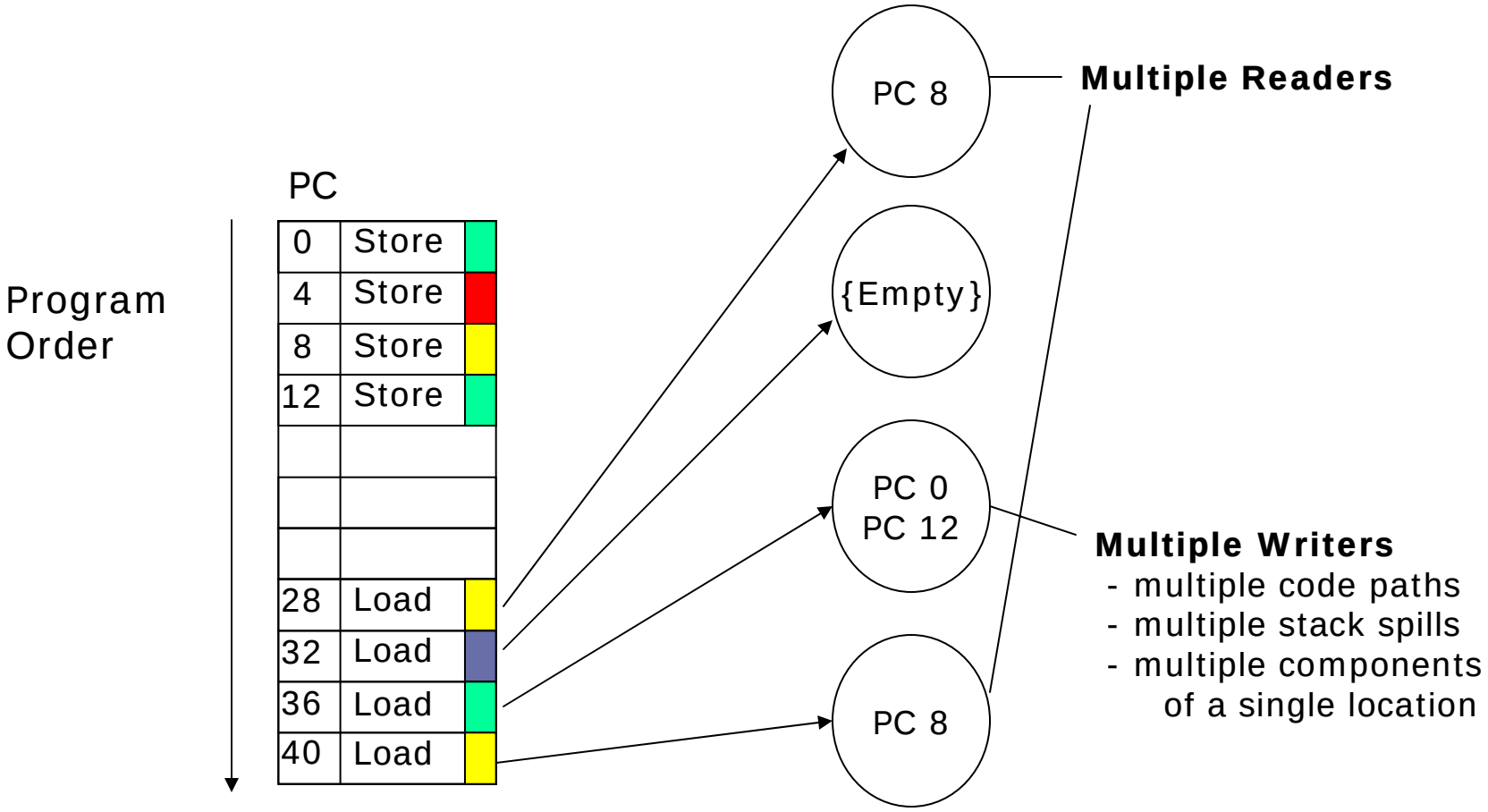
=> Large penalty for inaccurate address speculation

Memory Dependence Prediction
(Alpha 21264)

```
st r1, (r2)
ld r3, (r4)
```

- Guess that r4 != r2 and execute load before store
- If later find r4==r2, squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits

Store Sets
(Alpha 21464)

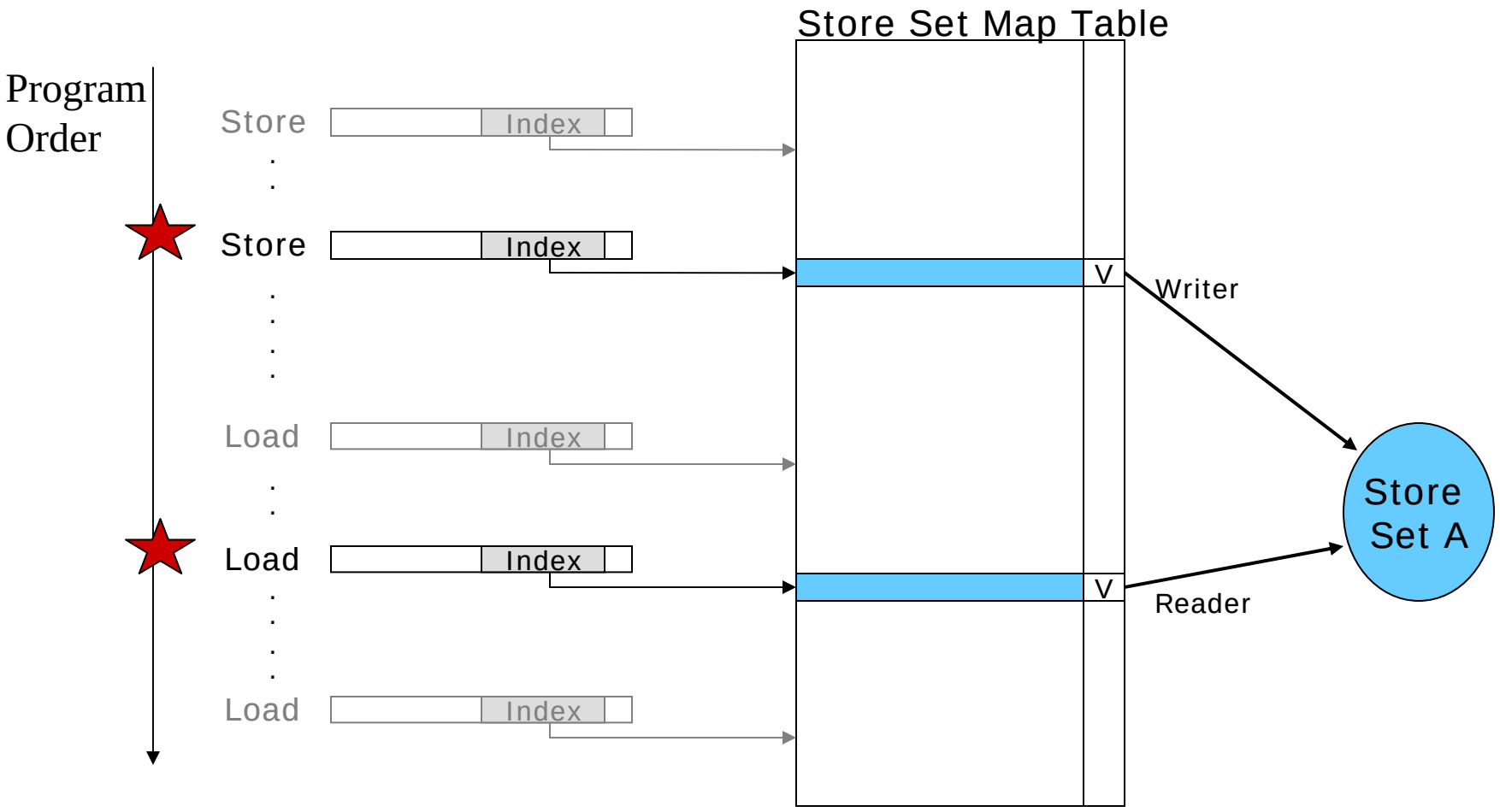


Memory Dependence Prediction using Store Sets

- The processor approximates each load's *store set* by initially allowing naïve speculation and recording memory-order violations.

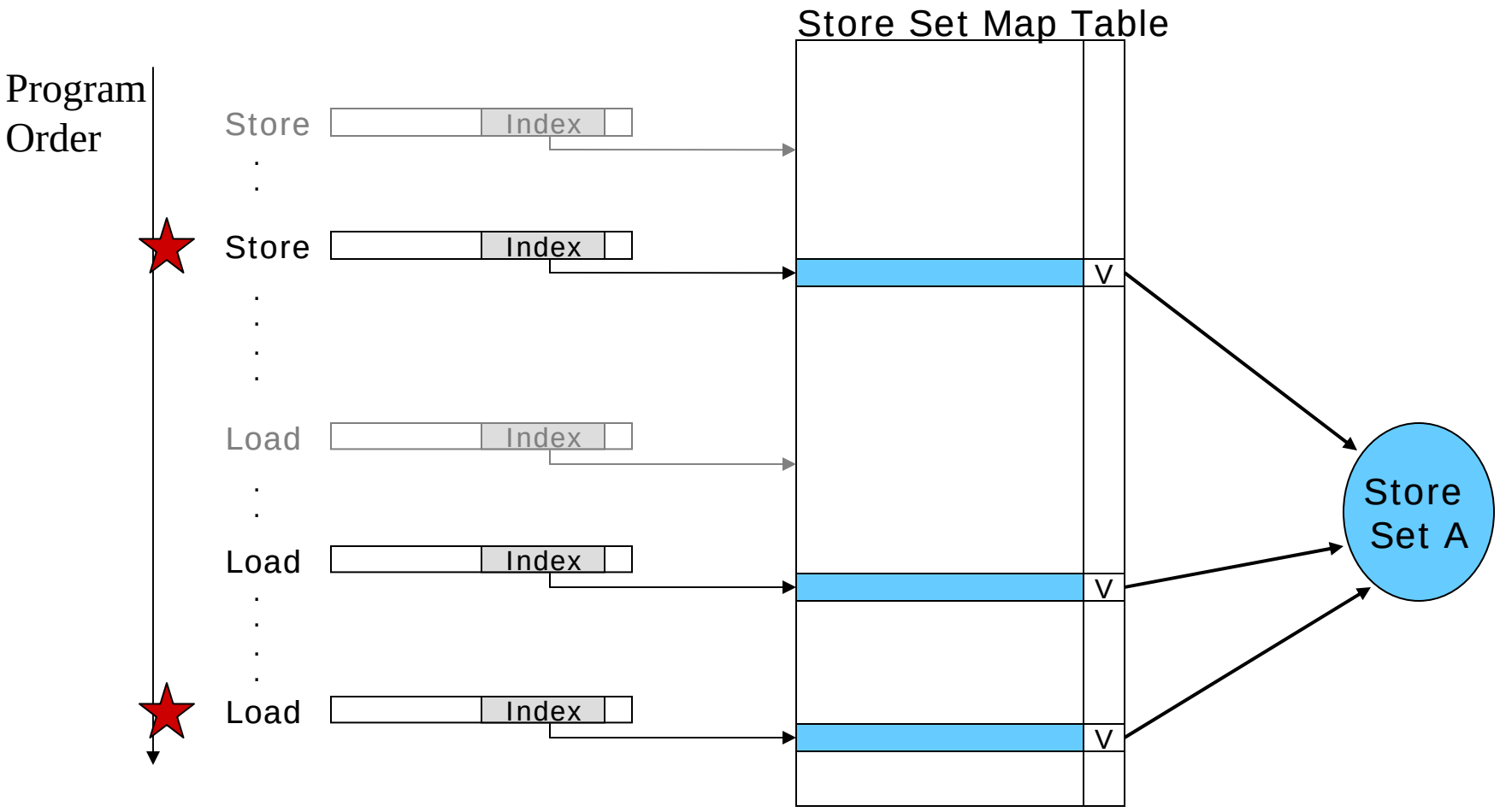
- A load must wait for any stores in its *store set* that have not yet executed.

The Store Set Map Table



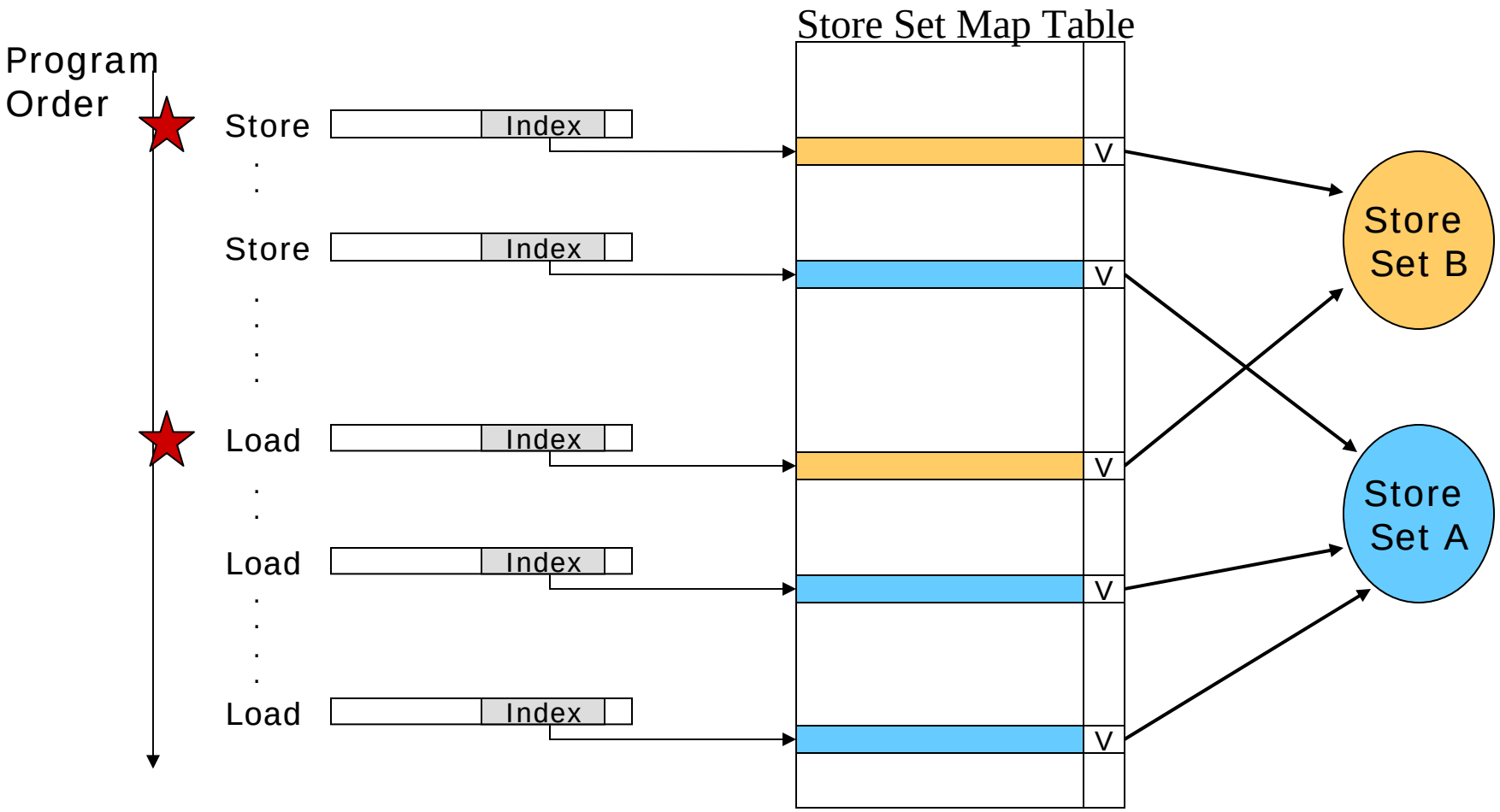
- Store/Load Pair causing Memory Order Violation

Store Set Sharing for Multiple Readers



- Store/Load Pair causing Memory Order Violation

Store Set Map Table, cont.



- Store/Load Pair causing Memory Order Violation

Thank you !

Extras

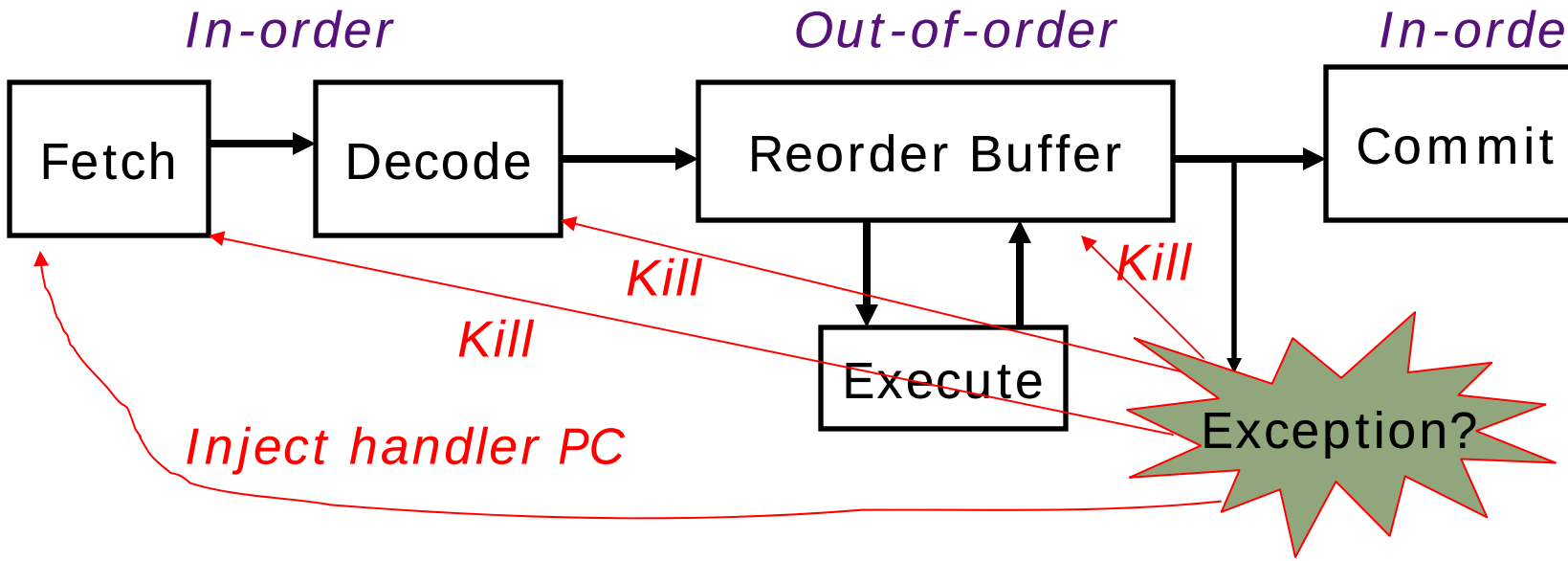
Mispredict Recovery

- In-order execution machines:
  - Assume no instruction issued after branch can write-back before branch resolves
  - Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

- Multiple instructions following branch in program order can complete before branch resolves

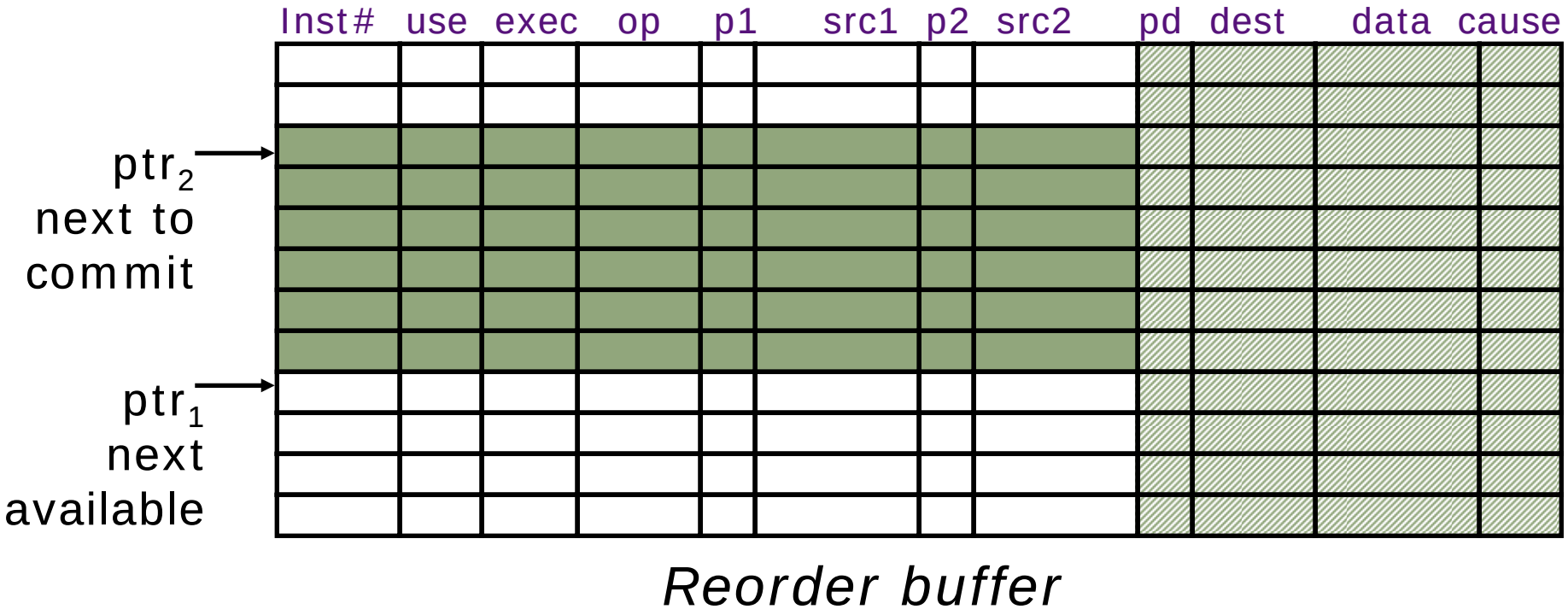
Precise Exceptions via In-Order Commit



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order

Temporary storage needed in ROB to hold results before commit

Extensions for Precise Exceptions

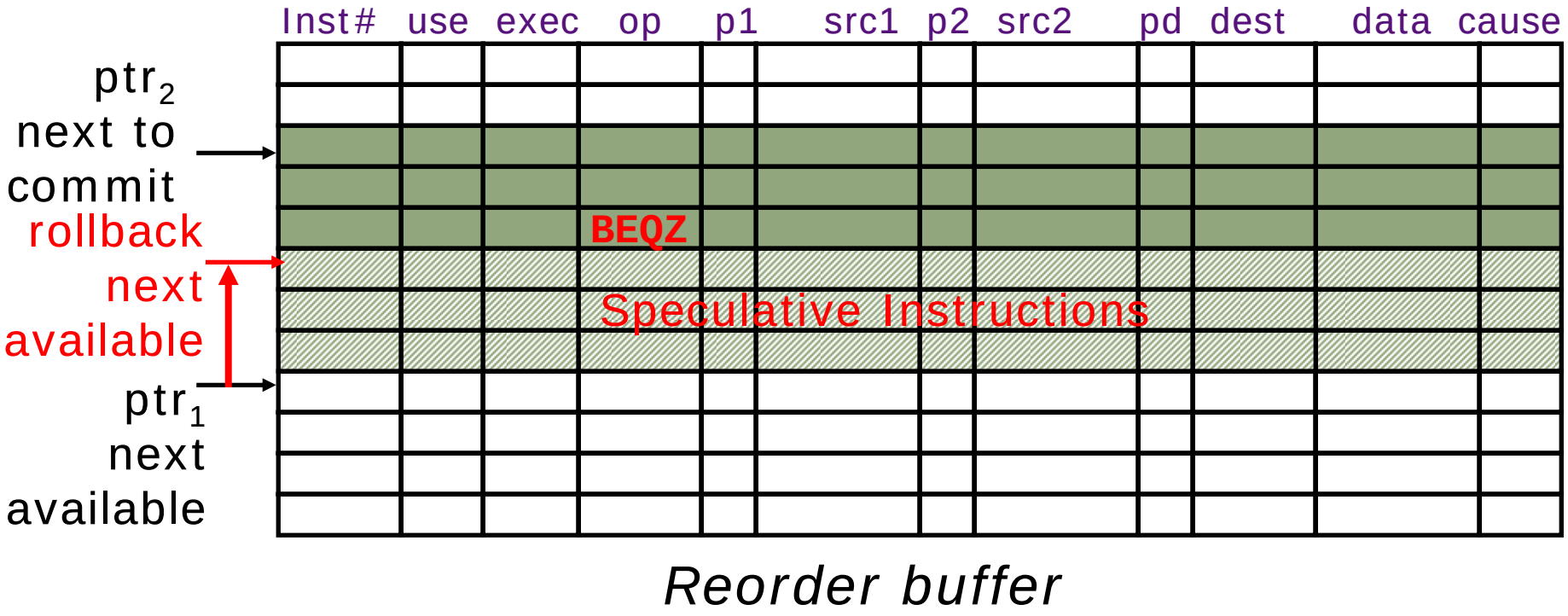


Reorder buffer

- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $ptr_1 = ptr_2$

  (stores must wait for commit before updating memory)

Branch Misprediction Recovery

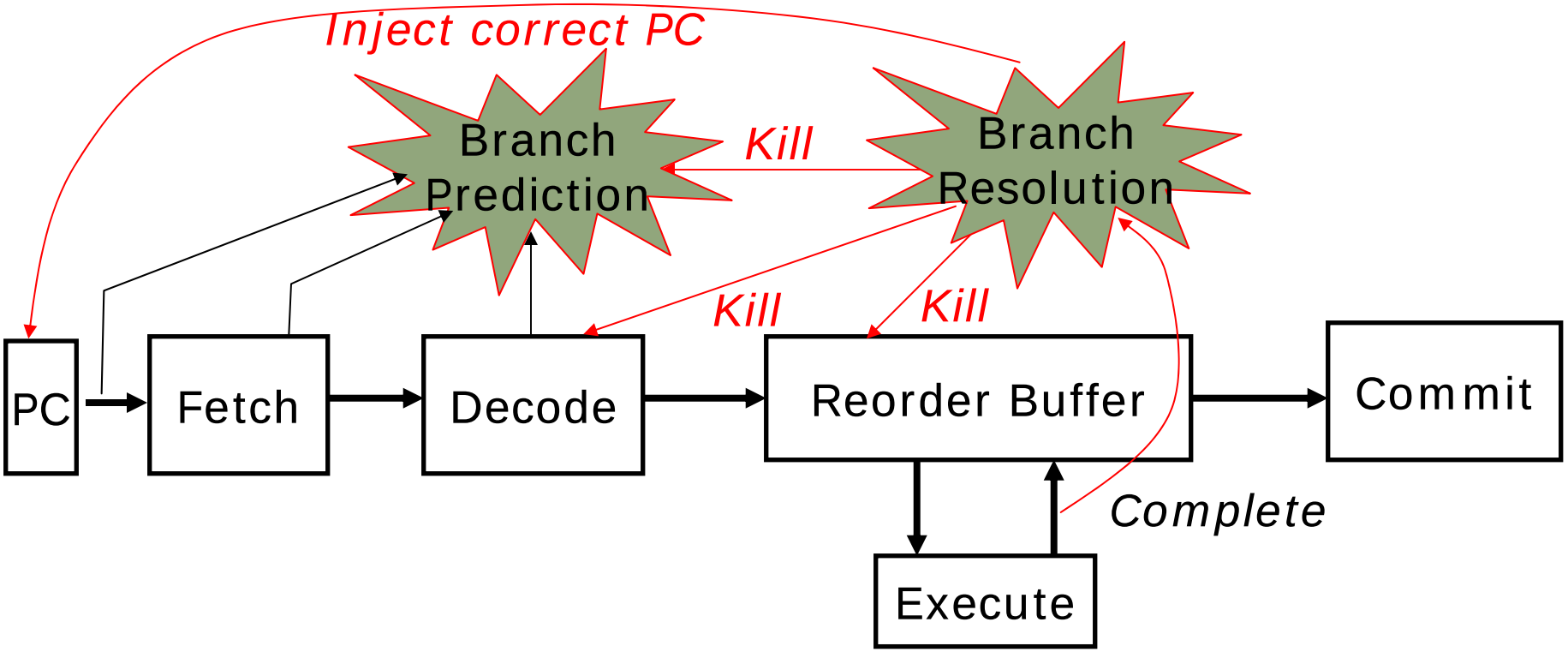


Reorder buffer

On mispredict

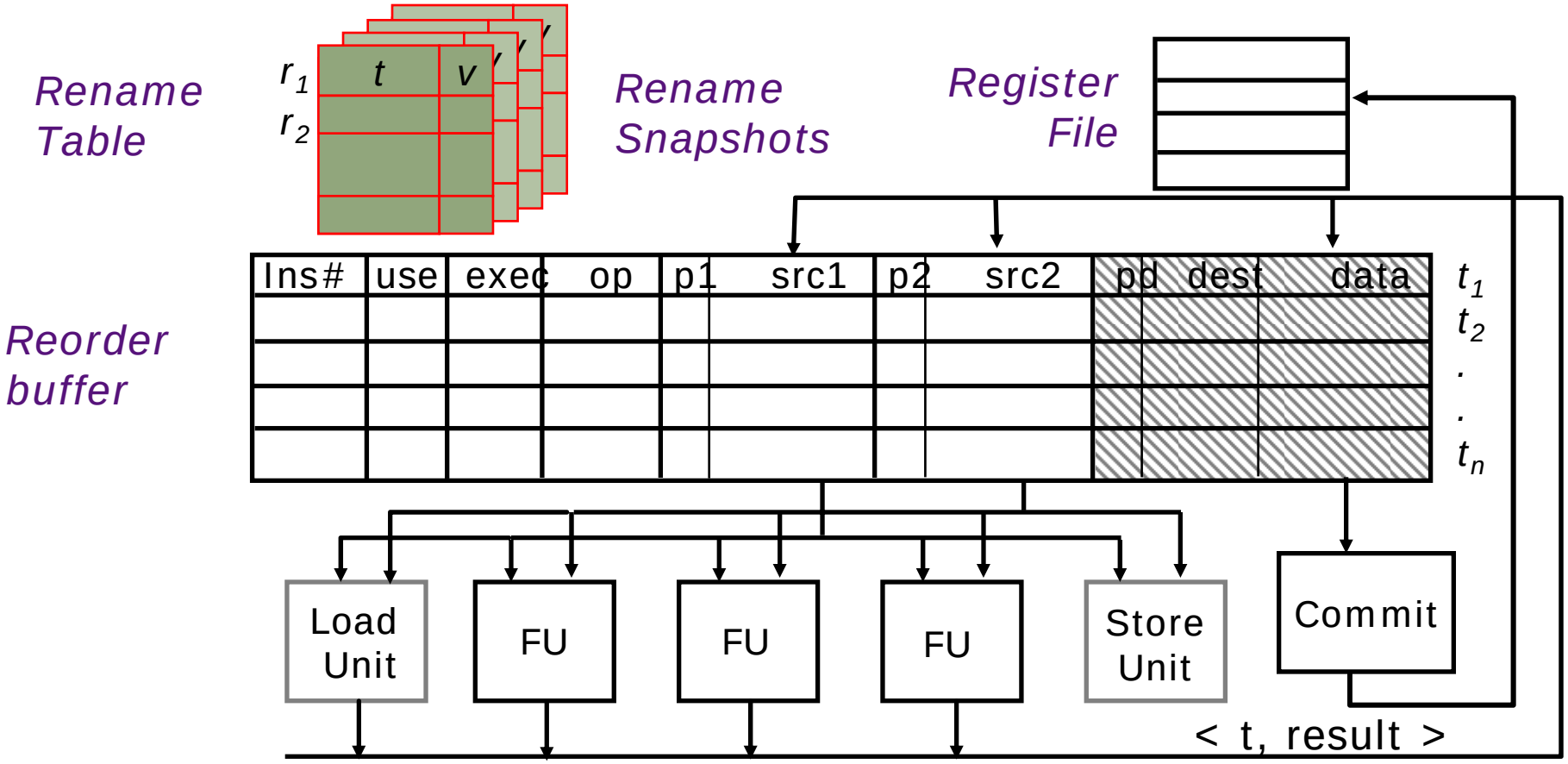
- Roll back “next available” pointer to just after branch
- Reset use bits
- Flush mis-speculated instructions from pipelines
- Restart fetch on correct branch path

Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

Recovering Renaming Table



Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted